6754

Richard B. Kalin
MIT Lincoln Laboratory
5 May 1971

Network Working Group
Request for Comments #150
NIC 6754

THE USE OF IPC FACILITIES

*** A WORKING PAPER ***

'This material has not been reviewed for public release and is intended only for use within the ARPA network. It should not be quoted or cited in any publication not related to the ARPA network.'

INTRODUCTION

It is our hypothesis that the goals of interprocess communication are more complex than commonly realized, and that until this complexity is more fully understood, there will be no satisfactory implementations. The objective of an IPC design must be more than that of providing a facility for moving bits between otherwise independent user programs, a variety of other constraints must also be satisfied.

These constraints are dictated by the eventual usage of the facility. Any design that will sustain this usage pattern can be a satisfactory one. If it does so efficiently, it will be said to be well designed. Furthermore, it is unimaginable that a large design effort, undertaken without a complete understanding of the usage it must serve, will ever be well designed or even satisfactorily designed.

This paper undertakes the exposition of the types of usage to which an IPC facility would be subjected, in hopes that such

a discussion will clarify the goals being pursued and will provide a benchmark for gauging various implementation strategies. The difficulty of this task should not be underestimated. The only experience available for us to draw upon is from very primitive and overly constrained IPC implementations. Extrapolation from this limited usage environment to more general notions has as yet no basis in real experience. Such speculation is therefore subject to enormous oversight and misguided perspective.

In compiling this paper, it was necessary to imagine what services a process might want from an IPC facility. The areas recognized include:

1) the exchange of bit encoded information via channels.
2) the establishment,deletion, and reassignment of these channels.
3) the ability to debug errors and suspected errors.
4) the potential to improve running efficiency.
5) the capacity to avoid storage allocation deadlocks.
6) the aided recovery from transmission errors.

This list is known to be incomplete. Some areas, such as 'accounting', have been ommitted because they are not well enough understood to be intelligently discussed. In other cases, ommissions should be blamed on simple oversight.

Because of these obvious problems, one should not consider any document of this kind as either authoritative or final. For this reason, this paper is being kept as a computer based textfile, and so will remain subject to editting and rerelease whenever new insights become understood. We hope that it can remain an accurate and up to date statement of the goals to which any group of IPC implementers can aspire and, as such, can be a guidebook to the problems that must be faced.

For several reasons no attempt shall be made here to design suitable solutions to the problems raised. To be

useful, such solutions must be machine dependent. A so called 'general solution' would actually be too clumsy and inefficient to be of any value. Secondly, we take the point of view of the user, who need not be aware of the manner in which his demands are carried out, so long as they can be accomplished. Finally, we are trying to stay aloof from the eccentricities of present day machine organization.

In our attempt to be implementation independent, we are admittedly treading a fuzzy line. Our characterization of usage patterns presumes both a system/process software organization and a computing milieu capable of supporting it. Although this does not appear to significantly affect the generality of the document, some care must be exercised in the selection of host machines.

*****

In the rest of this paper, we attempt to characterize the types of usage that should be anticipated by an IPC facility. The organization is into titled sections, each section discussing some aspect of the expected usage.

ABILITY TO USE VARIOUS SOURCES OF WAKEUP INFORMATION

Most processes exhibit preferences toward certain quantities of input data. This preference is reflected in the amount of computing time that can be expended for a given amount of input. For example, a character translation routine might prefer eight bit quantities of input. With seven or less bits no processing is possible, but once a complete character is available an entire translation cycle can commence. This preference is independent of the function of the routine.

Otherwise equivalent routines could be written that would accept one bit at a time. In other examples, a command interpreter might require a complete line of input, a linking loader a complete file.

Every executive system must face the problem of deciding at what times enough input is available for a given routine for it to run efficiently. This decision can not be taken lightly. Issuing a wakeup to a dormant process carries with it considerable overhead -- room in core storage must be made available, the program must be swapped into memory, tables must be updated, active registers exchanged, and the entire procedure done in reverse once the process has finished. To issue a wakeup when there is insufficient input for the program is inefficient and wasteful. The amount of computing that can be done does not justify the overhead that must be expended.

The problem of determining a priori the best time to issue a wakeup has no general solution. It depends critically upon the relationship between waiting costs and running costs. Attempts to make reasonable predictions must contend with the tradeoff between accuracy and overhead. The more system code that must be run, the more overhead incurred and the less the final prediction means.

Although there is no general solution, help is available to the scheduler in specific cases. A commonly found instance is that of using the receiving process to specify the number of bits that it is expecting. Thus, a process may inform the supervisor/scheduler that it requires fifty bits of input from some source before it is able to continue. The process can then go into the shade and it will be awakened when the required input is available.

In cases where input lengths are predetermined, this

technique is quite satisfactory. Elsewhere, problems arise. In the case of unknown input sizes, too short a prediction will give rise to the inefficiencies of premature scheduling, and too long a prediction may produce input deadlocks.

Under these circumstances it is common to have the process tell the scheduler a simple criterion that can be applied to determine if there is sufficient input -- the appearance of a carriage return in the teletype input stream, for example. The criterion is tested either by system routines or by a low overhead user supplied routine (which in turn must have a criterion of its own for being awakened). Once the criterion is met, the main routine is awakened and processing continues.

Sometimes the system and user criteria work in conjunction with one another. A user may specify a maximum character count, corresponding to available buffer size, and the system may look for line terminators. Wakeups to the user process may appear from either source. At other times the system may preempt the user's criterion. For example, if a process while trying to put a single character into a full buffer is forced into the shade, it will typically not be awakened again until buffer has been almost emptied. Even though the user program only wished room for a single character, the system imposed a much stronger criterion, namely room for N characters, on the assumption other calls to output characters will follow. Thus the program is forced into outputting in bursts and, rather than going into the shade and being awakened N times, each time when there is only room to output one character, the program is awakened once and sends N characters. Program efficiency is appropriately improved.

A third source of criteria for deciding when to awaken the user process is the device or routine that is producing the input data. This source is frequently utilized in hardware design. Many computer peripherals have the ability to send an

end of record indication. For variable length uninterpreted records this is an absolute necessity. For fixed length records it is a convenient double check. In the world of interprocess communication an analogous feature should be available. If the routine that is generating the data knows how much the receiving routine will require, then this information should be made available for scheduling purposes. Implementing this requires a standardized way of denoting logical boundaries on the stream of data flowing through a communication channel. The markers must be recognizable by the scheduler/supervisor in the receiving host computer so that wakeups can be issued as desired. To simplify the task of input interpretation, marker placement should also be visible to the receiving process.

The data between boundaries we shall call a logical message, since it is a natural unit of information exchange and since the markers travel with the data through the channel. The additional information the markers provide about data flow yield many useful consequences. Consider, for example, two processes that always exchange 100 bit long logical messages. If the receiving process is able to determine the length of each logical message that arrives, there is available a simple facility for error detection. If a 99 bit message arrives, it is obvious that a bit has been dropped somewhere.

It should be noted that it is not always possible for the receiving process to compute the positions of boundary markers in the input stream. There is no reason that the information implicit is marker position must also be found as part of the coded input data. Even in cases in which there is coding redundancy, it may be more difficult to extract boundary information from the rest of the input than it is to use the markers explicitly.

ABILITY TO SEND PARTS OF LOGICAL MESSAGES

Any IPC facility, in which user storage is at all constrained, can not require a process to send an entire logical message at one time. The concept is only introduced to facilitate the issuing of wakeups to a receiving process. What are convenient input quanta for the receiving process may not be convenient output quanta for the sending one. Consider the case of a process running on a small machine and sending messages to a process on a large task-multiplexed machine. For efficiency, the receiving process requires large quantities of input data at a time. Buffer space in the address space of the sending process can not hold much data. The only answer is to allow the sending process to dump its logical message in parts and with the last part to indicate that it is the end of a message.

ABILITY TO RECEIVE A LOGICAL MESSAGE IN PARTS

In the reverse of the above situation, a receiving process may not have sufficient buffer space available to accept an entire message at once. It should be possible under these circumstances to elect to accept the message in parts. This is also necessary in the case of messages that are too long to be buffered by the system. Unless part of the message is accepted by the receiving process, the transmission can never be completed. This device also serves for the removal of very long messages that appear by error in the input stream.

ABILITY TO FIND OUT IF A MESSAGE CAN BE SENT

If left unchecked, a routine can easily generate messages faster than they can be consumed. Since any given amount of buffer capacity can be quickly exhausted, there must be a way

for the system to limit the rate at which a process produces messages. This implies that at times a process trying to send a message may be prevented from doing so because of buffer inavailability. If the process is forced into the shade, the pause should not come without warning. There should be a way for the process to learn in advance if the message can be sent. A program may have better things to do than wait for a buffer to become available.

ABILITY TO GET A GUARANTEE OF OUTPUT BUFFER SPACE

A process should be able to get guarantee from the system that a message of a certain size can be sent. This allows the process to know before a computation is made that the results can be successfully output. This allocation should remain either until it is depleted or until it is changed by another allocation request.

This particular user option is sure to raise the wrath of legions of system programmers. From their point of view, the empty buffer space that is being preallocated is necessarily being wasted. For although it contains no messages, it is not available for other uses. The user, on the other hand, does not corrolate 'empty' with 'wasteful' nor 'full' with 'efficient'. A process needs empty output buffers as much as it needs full input ones. Both are resources necessary to sustain processing.

ABILITY TO FIND OUT ABOUT OUTSTANDING MESSAGES

A process that is sending messages over a channel should be able to find out how many of those messages have not yet been accepted by the user process at the far end and whether or not this number can decrease. Ideally, it should also be able

to determine the number of bits left in any partially accepted message, but the overhead necessary to implement this on conventional systems may be too great to be tolerated.

The count returned can be useful both dynamically and for post mortems. Used in debugging a remote process, it allows inputs on normally concurrent channels to be presented one at a time and in any given order. In this way one could, for example, verify that the same results are produced regardless of the order in which the inputs arrive.

If there is a failure of a receiving process, this mechanism allows a sending process to determine the last input accepted before the process died. Even between operational processes it provides a user managed way for the transmitting process to slow down and let the receiver catch up with it. By pinpointing bottlenecks, it can be used to detect design mismatches.

Unless the channel has no outstanding messages or it is dead, there is the possibility that concurrently with the request the receiving process will accept another message. This being the case, the count returned can not be assumed to be exact but must be considered as only an upper bound.

ABILITY TO GET WAKEUPS WHEN MESSAGES ARE ACCEPTED

In conjunction with the above it should be possible for a user process to be alerted when the number of messages that have been sent over a particular channel and not accepted at the far end falls below a specified threshhold. Thus a process, upon discovering that twenty messages are still outstanding, can elect to enter the shade until this number has fallen to less than five. By doing so the process can run in 'burst mode'. Rather than being swapped in and out of core

fifteen times and each time being allowed to send one message, it is loaded once and sends fifteen messages. There is no penality for doing this since the bottleneck on throughput is at the receiving process. If swapping costs for the local process are significant, there may be considerable economic advantage to this mode of operation.

If the remote process dies or issues a channel 'close', the count of undelivered messages becomes frozen. If the receiving process is expecting this type of wakeup, it should get one at this time even though the count has not reached the desired threshhold. The process is thus alerted to do a postmortem on the channel.

ABILITY TO LEARN ABOUT MESSAGES QUEUED FOR INPUT

A process should be able to learn of the status of the Nth logical message queued on a given input channel. It should a least be able to determine if it is available, whether or not it is complete, how long it is and what it contains.

This facility allows a program to make general preparations before accepting a message. It offers some escape from being put into the awkward position of having accepted input and not being able to dispose of it. If for example, it is known that processing the message will result in two more messages being sent, then it is advantageous to get guarantees that the output can be generated before the input is accepted.

Under circumstances in which one end of a channel is moved from one process to another, for example, moving a teletype connection between a user program and a debugging program, this ability to scan ahead in the input stream allows a process to check whether or not pending input is really meant for it. If it is, the input will then be accepted normally, otherwise, the

end of the channel must be first moved to another receiving process.

Accepting input should be viewed as a grave responsibility, not to be undertaken unless there is reasonable assurance that the input can be processed. One of the first rules of asynchronous system design is to detect errors as soon as possible. If propogated, the tangled results may be hopeless to unravel.

ABILITY TO LEARN HOW MANY MESSAGES ARE WAITING

A process should be able to determine how many messages are left to be processed on a given input channel. Two uses are readily thought of. Given pending inputs on several channels a process should be able to exercise preference. One decision might be to take input first from the channel with the most messages queued. This might have the effect of increasing throughput since by freeing message buffers the remote transmitting process might be allowed to run. Another possibility might be that the receiving process has some control over the sending process and, upon observing the backlog on inputs, it could tell that process to slow down.

Assuming that the remote process is still able to send messages, the number of inputs reported is only a lower bound. New inputs may be added concurrently. If the foreign process has died or has otherwise closed the connection then the bound can be made exact. The local process should be able to learn when it is exact.

GUARANTEE THAT INPUT WILL STAY AVAILABLE

This requirement states that if a process has been told

that it is able to receive N messages on a given channel, that those messages are really available and buffered within the host machine. If promised to a user process, messages should not mysteriously become unavailable. An example of how this might happen is illustrated in RFC.60. There, during a panic for buffer space, messages are destroyed and reported as being in error. They are later recovered from backup copies contained in the foreign host.

ABILITY TO RECEIVE A WAKEUP WHEN INPUTS ARRIVE

A process should be able to enable a wakeup when the number of messages queued on an input channel exceeds a specified value or has reached its maximum value. This allows a program to process input in a burst mode fashion and to economize on swapping costs. It also permits inputs to be combined in a simple manner. If, for example, two inputs are needed before anything can be done, then the appropriate interrupt can be easily generated.

The same interrupt should be generated if the maximum number of inputs have been received. Two cases are distinguished. Either the foreign process has closed the channel and is therefore not sending more messages, or the system will not allocate more buffers until some input is accepted. In this way the process can be informed that there is no point in waiting for the condition it anticipates.

ABILITY TO SPECIFY SPECIAL WAKEUPS

A process, when trying to run efficiently, should be able to specify arbitrarily complicated wakeup conditions. This allows a user managed way of minimizing the number of premature wakeups. This generailty is perhaps most easily provided for

by allowing the main process to designate a small low overhead interrupt driven routine that will check for the desired conditions and issue a wakeup to the main process whenever they are met.

ABILITY TO MEASURE CHANNEL CAPACITY

There has been much discussion about the measure of a datastream and in the heat of committee, much confusion has been generated. It is our feeling that, within the present domain of discussion, there is no single measure of the capacity of a message channel. Two completely orthogonal concepts must be measured -- 1) the number of messages buffered and 2) the number of bits of encoded data represented. The system overhead associated with each is very much implementation dependent and hence no general equation can express the measure of one in terms of the other. By making an arbitrary assumption (eg. a message boundary equals 100 bits of buffer), a system runs the risk of excluding new nodes that are unable to meet the old criterion.

ABILITY TO FIND OUT MAXIMUM CHANNEL CAPACITY

There should be provided a system call that enables a user process to learn of the maximum current capacity of any given channel. This should be reported as a pair of numbers, namely the maximum bit count and the maximum message count.

ABILITY TO CONSTRAIN CHANNEL CAPACITY

A process using a channel should be able to set new bounds on the capacity of a given channel. If possible the system should try to meet this bound. It should be noted that the actual bounds imposed must meet the constraints of at least

four different sources -- local and remote user process, local and remote system -- by setting a arbitrarilly high bound, no guarantee is made that it can be met. Similarly, a low bound can not always be met until buffered messages are consumed.

Thus a receiving process, by setting the current message bound to zero, effectively disables the transmission of new messages. Thus, without the cooperation of the transmitting process, message generation can be temporarilly disabled, while outstanding message buffers are flushed. Later the message allocation can be raised to its original limit and transmission can be resumed.

ABILITY TO CLOSE A CHANNEL AT ANY TIME

A process should be able to close down a channel at any time. If the process has died, the system should be able to close all open channels for it. For channels over which the process was receiving data, pending input should be thrown away and indications returned to the transmitting system marking the channel as dead and identifying the last data item accepted. This identification will be in terms of the number of logical messages discarded and the number of bits left in the oldest message.

If a process closes a channel over which it had been sending, buffered output should be sent normally, and with it should be sent an indication that this is all of the input that will ever appear.

ABILITY TO GIVE AWAY CHANNEL PRIVILEGES

The right to perform any of the operations discussed here is normally reserved by the process that established the

channel. At times that process may wish to transfer some of its delegated power to another process, especially in an environment where one process may spawn another and resources must be passed to the newly created process.

Schemes for such reassignment can become arbitrarilly complicated. One could, for example, assign each of the various aspects of usage individually and then separately assign the various rights of reassignment. Fortunately it is not always necessary that it become so elaborate, it is expected that in most cases the following simple strategy can suffice. The ability to close a channel is retained exclusively by the process that established the channel. If the channel is still open when the process dies, it is automatically closed by the system. All other uses of the channel remain outside system control. The channel is known by name and all processes to which the name has been given may make use of the channel. It is left to user level coordination to insure that only one process is actually making use of it at any one time.

ABILITY TO INITIATE CHANNEL CREATION

For most cases channel establishment can be handled quite simply. a process announces to its local system that it listening on a specified channel. It is connected to the first remote process that 'dials' the right number. Identification of the caller takes place only after the channel has been established. In the event of a wrong number, the channel can be closed and the listening resumed. Callers trying to reach preestablished channels will get a 'busy signal'.

To 'dial' a remote process a process must specify a channel on which it is listening and a remote number. The system will then attempt to establish the connection. The

channel will become 'busy' during this time.

For processes that prefer to avoid the complications of identifying remote callers, an additional feature can be added. By specifying both the local and remote channel identifiers a process can transfer to the system the responsibility for screening callers for the proper identification. The connection will only be accepted from the caller specified.

ABILITY TO REPORT TRANSMISSION ERRORS

If after prescanning an input message a process should decide that it contains some sort of transmission error, it should be able to reject the message. The system should then invoke any internal error recovery mechanism that it may have implemented.

*****

POSTSCRIPT

The author welcomes any comments, questions, or corrections to this document. Even the most informal note or telephone call will be appreciated. Especially of interest are opinions about the usefulness of the discussion and whether or not there should be more papers directed at other of the basic questions of computer networking. If the consensus tends to the affirmative, then others are encouraged to contribute working papers on the problems of flow control, error handling, process ownership, accounting, resource control, and the like.

RBK/TX?